1 Modello astratto di calcolo

Al fine di dare una valutazione quantitativa della complessità di un algoritmo che non dipenda da una particolare piattaforma Hw/Sw si confrontano gli algoritmi rispetto a un modello teorico di calcolo che definisce un insieme di operazioni elementari e il loro costo. Nel nostro caso, cominciamo col considerare le seguenti operazioni di base, alle quali assegniamo costo unitario:

- Assegnzioni.
- Operazioni aritmetiche; ad esempio, $A = B*C + D \;\, 7$ ha costo 4, (3 operazioni aritmetiche più un'assegnazione).
- Test; ad esempio, `if` $(A == B)$ ha costo 1.
- Lettura/Scrittura; ad esempio, `System.out.println(A)` ha costo 1.

A partire da queste, si definiscono di conseguenza i costi di operazioni complesse.

- Ciclo: il costo complessivo è dato dalla somma del costo del test e del costo del corpo del ciclo, ciascuno pesato per il numero di volte che viene eseguito.
- `if` ...`then` ...`else` : il costo è quello del test più quello del blocco di istruzioni che viene eseguito.
- Attivazione di un metodo: il costo è dato dalla somma di tutte le istruzioni del metodo eseguite.

Si osservi che:

1. Il costo di istruzioni complesse dipende dal particolare input considerato. Ad esempio, il costo di un'istruzione `if` ...`else` dipende da quale blocco di istruzioni è eseguito e questo a sua volta è determinato dall'esito del test e dunque, in definitiva, dal particolare input.
2. L'insieme di operazioni

Il costo complessivo per un particolare input è dato dalla somma di tutti i costi.

Il modello sopra descritto può permettere di valutare quantitativamente gli algoritmi, in modo indipendente dalla piattaforma Hw/Sw. È lecito chiedersi quale sia la relazione tra il costo valutato usando il modello sopra descritto e quello che si otterrebbe eseguendo una implementazione dell'algoritmo su una particolare piattaforma. Osserviamo innanzi tutto che valgono le considerazioni seguenti:

1. Ogni istruzione elementare corrisponde ad un numero finito (e limitato da una costante) di istruzioni macchina su qualunque piattaforma.
2. Il costo di due istruzioni macchina su una qualunque piattaforma è sempre legato da una costante; ad esempio, consideriamo un processore che implementi le due operazioni `ADD A, B` (addizione) e `MUL A, B` (moltiplicazione), dove `A` e `B` sono due interi rappresentati in complemento a 2 con 32 bit. Le due operazioni sono legate da una costante. Ad esempio, la prima richiede 10 ns (nanosecondi), la seconda 125 ns. In questo caso la costante che lega le due operazioni è 12,5.
3. I tempi di accesso a variabili in memoria principale sono legati da una costante; ad esempio, i tempi di accesso ad una variabile di tipo `long` e ad una di tipo `double` saranno in generale diversi, ma differiranno al massimo per una costante che dipende dal sistema Hw/Sw in considerazione.

Chiariamo i concetti sopra esposti considerando il seguente algoritmo per il calcolo del massimo in un array di n elementi, implementato come una funzione C:

```
void arrayMax(int A[], n) {
      int i;
      int currentMax=A[0];
         for (i=0; i < n; i++)
            if (A[i] > currentMax)
               currentMax=A[i];
      return;
}
```

Consideriamo il seguente vettore di input: {7, 1, 4}. È facile verificare che per tale input particolare il costo secondo il modello di calcolo definito sopra è di 13 unità. In che relazione è tale misura con l'effettivo tempo di esecuzione del programma su una particolare piattaforma Hw/Sw? Supponiamo di considerare una piattaforma particolare e di aver stabilito (ad esempio mediante misurazioni sperimentali) che su tale piattaforma i tempi di esecuzione delle operazioni che ci interessano sono i seguenti:

- Assegnazione di una variabile intera: 20 ns.
- Test: 10 ns.
- Incremento di una variabile intera (ad esempio `i` nel ciclo `for`): 30 ns.
- Attivazione della funzione (allocazione della memoria per le variabili locali, creazione del record di attivazione ecc.): 50 ns.
- Esecuzione dell'istruzione `return`: 30 ns.

Ponendo convenzionalmente pari ad 1 il costo dell'operazione di test, le operazioni hanno costi 2 (assegnazione), 1 (test), 3 (incremento), 5 (attivazione), 3 (`return`). Ricalcolando il costo dell'esecuzione usando questi pesi otteniamo il valore 2 (assegnazione iniziale) + 2 (assegnazione nel ciclo `for`) + 3 (test nel ciclo `for`) + 9 (incrementi nel `for`) + 1 (test finale `for`) + 3 (test `if`) + 3 (istruzione `return`) = 23. Il rapporto tra il costo calcolato in quesdto modo e quello valutato usando il modello definito sopra è circa 1,77.

È opportuno sottolineare che i due costi differiscono per i seguenti motivi (a parte la scelta dell'unità di misura, che può essere fatta arbitrariamente):

- Il costo di un'operazione elementare non è quello stabilito nel modello teorico (a parte le unità di misura); ad esempio, il costo dell'incremento è stato posto pari ad 1 nel modello, ma risulta essere di 3 unità in pratica.
- I costi relativi delle operazioni sono diversi nei due casi; ad esempio, nel modello teorico l'operazione di test e quella di assegnazione hanno entrambe costo unitario, mentre in pratica l'istruzione di assegnazione costa il doppio dell'operazione di test. Si osservi tuttavia che i due costi sono legati da una costante (2 nel caso particolare considerato).
- Nel modello teorico non si considerano i costi di attivazione delle funzioni (o dei metodi, nel caso di linguaggi Object Oriented). A tale riguardo si osservi tuttavia che, essendo l'attivazione eseguita una sola volta per l'esecuzione di una funzione (o di un metodo), tale differenza incide solamente per un fattore costante nella stime del costo di esecuzione.

Queste considerazioni ci portano a concludere che *il costo dell'algoritmo valutato secondo il modello teorico e il tempo di esecuzione dell'algoritmo su una piattaforma Hw/Sw reale sono legati da un fattore costante (a parità di input considerato).*

Per avere una valutazione che non dipenda dal particolare input, ma *solamente dalla sua dimensione*, si ricorre all'analisi del caso peggiore.

2 Primi esempi sulla complessità

2.1 Analisi dell'algoritmo `InsertionSort`

L'algoritmo è il seguente:

```
Algorithm InsertionSort(A, n)
for (i=0; i<n; i++) {
    tmp=A[i];
    for (j=i; j>0 && tmp<A[j-1]; j–)
        A[j]=A[j-1]; (1)
    A[j] = tmp;
}
return A;
```

Si osservi che il ciclo `for` più esterno viene comunque eseguito n volte, con il variare di i da 0 a $n-1$. Sia $C(i)$ il costo della i-esima esecuzione del ciclo `for` più esterno. Il costo dell'algoritmo è allora $O(\sum_{i=0}^{n-1} C(i))$. Consideriamo la generica iterazione (la $(i+1)$-esima) del ciclo `for` più esterno. Se analiziamo la struttura del ciclo `for` più interno ci rendiamo conto che nel caso peggiore l'elemento A[i] viene confrontato (e scambiato) con tutti gli i elementi precedenti (A[0],...,A[i-1]). Ciò significa che nel caso peggiore l'istruzione (1) è eseguita i volte durante la $(i+1)$-esima iterazione del ciclo `for` più esterno. D'altra parte, $C(i)$ è dato proprio dal numero di volte che (1) è eseguita, a parte una costante, dunque $C(i) \leq c \cdot i$, dove c è la costante suddetta. Ciò significa che il costo dell'algoritmo nel caso peggiore è $O(\sum_{i=0}^{n-1} i)$. Infine:

$$\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2} = O(n^2).$$

Un modo semplice per avere lo stesso risultato (asintoticamente) è osservare che il ciclo esterno è eseguito al più n volte e cosi' pure quello interno. Questo modo di procedere meno accurato non peggiora gravemente l'analisi dell' `InsertionSort`, ma può portare a sovrastimare eccessivamente la complessità in altri casi.

2.2 Analisi dell'algoritmo `MergeSort`

L'algoritmo (descritto come metodo Java) è il seguente:

```
void mergesort(int[] A, int first, int last) {
    if (first < last) {
      int mid = (first + last) / 2;
      mergesort(A, first, mid);
      mergesort(A, mid+1, last);
      merge(A, first, last);
    }
}
```

L'algoritmo è eseguito su un array `A` invocando il metodo seguente:

```
void mergesort(int[] A) {
    mergesort(A, 0, A.length-1);
}
```

Ai fini dell'analisi, non è necessario dare qui i dettagli del metodo `merge`. È solo importante ricordare che, dati due array singolarmente ordinati e di dimensioni rispettive n_1 ed n_2, `merge` li fonde in un nuovo array (ordinato) di dimensione $n_1 + n_2$ in tempo $O(n_1 + n_2)$ e dunque $c_1(n_1 + n_2)$ per un'opportuna costante c_1. Supponiamo che l'array `A` contenga n elementi. Indichiamo con $C(n)$ il costo del `mergesort` nel caso peggiore per un array di n elementi, dove n è un intero qualsiasi. Il costo dell'algoritmo descritto sopra può allora essere limitato nel modo seguente:

$$C(n) \leq \begin{cases} c_2, \ n = 1 \\ C\left(\left\lceil \frac{n}{2} \right\rceil\right) + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c_1 n, \ n > 1 \end{cases}$$

Si osservi che il costo c_2 quando $n = 1$ è motivato dal fatto che almeno l'istruzione `if` è eseguita, mentre, per $n > 1$, il costo dell'istruzione `if` e della successiva assegnazione della variabile `mid` è costante e può essere conglobato nel termine $c_1 n$, che a sua volta dà il costo del `merge`. Inoltre, nel seguito trascuriamo il fatto che n è intero e dunque non consideriamo le parti intere inferiori e superiori nelle divisioni. Non diamo qui una prova rigorosa del fatto che tale approssimazione è lecita. Diciamo solo che essa è conseguenza del fatto che, per qualunque intero k, $C(k)$, $C(k-1)$ e $C(k+1)$ differiscono per una costante additiva. Dunque possiamo scrivere:

$$C(n) \leq \begin{cases} c_2, \ n = 1 \\ C\left(\frac{n}{2}\right) + C\left(\frac{n}{2}\right) + c_1 n, \ n > 1 \end{cases}$$

Cerchiamo ora di verificare che $C(n) = O(n \log n)$. Supponiamo inizialmente che $n > 1$. Dobbiamo dimostrare che $\exists c$ tale che $C(n) \leq cn \log n$ per n abbastanza grande. Nel seguito consideriamo $n > 1$. Il motivo di questa scelta sta nel fatto che se $n = 1$ allora $n \log n = 0$, mentre $C(1) = c_2 > 0$. Quindi, la nostra limitazione superiore di $C(n)$ certamente non vale per $n = 1$. Assumendo $n > 1$ e $C(n) \leq cn \log n$ possiamo scrivere:

$$C(n) \leq 2C\left(\frac{n}{2}\right) + c_1 n \overset{(1)}{\leq} cn \log \frac{n}{2} + c_1 n = cn \log n - cn + c_1 n,$$

dove (1) segue dal fatto che, per ipotesi, $C(k) \leq ck \log k$ per ogni $k > 1$ (e dunque anche per $k = n/2$ se n è abbastanza grande). Osserviamo che c_1 è una costante data, che dipende dall'algoritmo, mentre c può essere fissata a piacere *purché costante*. Scegliendo $c \geq c_1$, le diseguaglianze mostrate sopra implicano che $C(n) \leq c \log n$. Come osservato sopra, tale risultato vale quando $n > 1$, ossia per un valore sufficientemente alto di n, in accordo con la definizione della notazione O. Se vogliamo un'espressione che valga anche quando $n = 1$ possiamo osservare che $C(n) \leq n \log n$ implica che $C(n) \leq n \log n + b$ per qualunque costante $b > 0$. Scegliendo $b \geq c_2$ abbiamo $C(n) \leq n \log n + b$, e questo risultato vale anche per $n = 1$.

3 Esempi

1. Si hanno a disposizione una classe `Key` che implementa l'interfaccia `Comparable` e una classe `Heap` che mette a disposizione i metodi seguenti (per semplicità si suppone che le chiavi coincidano con gli elementi dell'heap):

- `public Key getMax()`: restituisce la chiave massima.
- `public Key delMax()`: restituisce ed elimina la chiave massima.
- Costruttore `public Heap()`: costruisce un heap vuoto.

Scrivere una classe `SuperHeap` che, oltre ai precedenti, metta a disposizione il metodo seguente:

- `public Heap firstkMax(int k)`: restituisce un oggetto di classe `Heap` contenente i primi k elementi a priorità maggiore.

Si supponga di avere a disposizione una classe `List` che mette a disposizione le consuete operazioni (e i corrispondenti metodi) `cons, cdr, car, isEmpty`.

Soluzione. Un metodo che risolve il problema può essere il seguente:

```
public class SuperHeap extends Heap {
      /* Costruttori ecc...  */
      public Heap fisrtkMax(int k) {
      List lista = new List();
      Heap myHeap = new Heap();
                for (i=0; i<k; i++) {
                    lista.cons(this.getMax());
                    myHeap.insert(this.delMax());
                }
                while (!lista.isEmpty())
                    this.insert(lista.cdr());
                return myHeap;
      }
      /* Altri metodi della classe SuperHeap ....  */
}
```

2. Calcolare la complessità nel caso peggiore della ricerca in un BST che gode della proprietà seguente: considerato qualsiasi nodo v e indicati rispettivamente con $n_l(v)$ ed $n_r(v)$ il numero di nodi dei sottoalberi sinistro e destro e posto $n(v) = n_l(v) + n_r(v)$, si ha:

$$\min\{n_l(v), n_r(v)\} \geq \frac{1}{3}n(v).$$

Si supponga che, nel caso peggiore, il sottoalbero con altezza maggiore sia quello con il maggior numero di nodi. Si indichi con n il numero complessivo di nodi dell'albero.

Soluzione. La complessità della ricerca nel caso peggiore è O(altezza dell'albero), quindi occorre limitare superiormente quest'ultima. Sia $h(n)$ l'altezza nel caso peggiore per un BST che goda della proprietà descritta sopra. Partendo dalla radice e considerando i sottoalberi sinistro e destro possiamo scrivere:

$$h(n) \leq 1 + \max\{n_l(radice), n_r(radice)\} \leq 1 + h\left(\frac{2n}{3}\right).$$

La prima diseguaglianza segue dal fatto che l'altezza dell'albero è al più pari alla profondità della radice aumentata dell'altezza del sottoalbero avente profondità massima. La seconda diseguagliaza segue osservando che: i) per l'ipotesi fatta sopra, nel caso peggiore il sottoalbero con altezza massima è quello con il maggior numero di nodi; ii) il sottoalbero con il numero maggiore di nodi ha al più $(2/3)n$ nodi, per l'ipotesi di bilanciamento.

Supponiamo che l'altezza di un albero che goda delle proprietà sopra descritte sia $a \log n + b = O(\log n)$ nel caso peggiore. Verifichiamo la correttezza di tale ipotesi.

$$h(n) \leq 1 + h\left(\frac{2n}{3}\right) \leq 1 + a \log \frac{2n}{3} + b = 1 + a \log \frac{2}{3} + a \log n + b,$$

dove la seconda diseguaglianza è vera per ipotesi. L'espressione sopra è limitata superiormente da $a \log n + b$ quando $1 + a \log(2/3) \leq 0$, che vale non appena

$$a \geq \frac{1}{\log \frac{3}{2}}.$$